DEVOPS E INTEGRAÇÃO CONTÍNUA

AULA 3

Prof. Mauricio Antonio Ferste

CONVERSA INICIAL

Tempos modernos, dirigido por Charlie Chaplin, é uma sátira icônica sobre a desumanização e as complexidades da vida industrializada durante a Revolução Industrial. Embora o filme seja uma obra-prima do cinema mudo, pode-se encontrar algumas analogias interessantes entre os temas abordados por Chaplin e os princípios do movimento DevOps na era da tecnologia moderna.

Figura 1 – *Tempos modernos*, de Charles Chaplin

Crédito: ALLSTAR PICTURE LIBRARY/Alamy/Fotoarena.

Embora as comparações entre um filme clássico e um movimento contemporâneo possam parecer distantes, ambos exploram questões fundamentais relacionadas a trabalho, tecnologia e humanidade em seus respectivos contextos.

Em *Tempos modernos*, é destacada a monotonia e desumanização do trabalho industrial repetitivo. A linha de montagem representa uma forma extrema de trabalho mecanizado. Em contrapartida, em DevOps, essa automação é crucial para eliminar tarefas manuais repetitivas. A padronização de processos garante consistência e eficiência. O filme destaca a falta de comunicação entre trabalhadores e a desconexão social resultante. Já em CI/CD temos ter uma estreita colaboração entre equipes de desenvolvimento e operações para melhorar a comunicação e a eficiência ao longo do ciclo de

vida do desenvolvimento. Em termos de velocidade, existe uma ênfase na produção em massa e na eficiência máxima no filme, que também é vista em CI/CD como uma busca por entrega rápida e eficiência, integrando desenvolvimento e operações para acelerar o ciclo de vida do *software*.

Principalmente em termos de adoção de tecnologia, a máquina de alimentação automática destaca a rápida adoção de tecnologia, como também ressalta suas implicações negativas. Em termos de DevOps, existe o envolvimento contínuo com novas tecnologias para melhorar processos, embora com foco na compreensão dos impactos e na mitigação de riscos.

Finalmente, em termos de resiliência e adaptação, em *Tempos modernos*, o personagem de Chaplin luta para se adaptar a um ambiente industrial em constante mudança. O mesmo em DevOps, o que destaca a importância da resiliência e capacidade de adaptação em um cenário tecnológico dinâmico.

O mundo se repete, temos de interpretá-lo.

TEMA 1 – INTEGRAÇÃO CONTÍNUA (CI) EXPLICADA

Integração contínua (CI, do inglês *Continuous Integration*) no Git é a integração contínua de código usando o Git, um sistema de controle de versão distribuído. A CI automatiza o processo de integração de alterações de código de vários desenvolvedores em um único repositório central.

1.1 Quando usar

Adotar as recomendações de práticas de engenharia do DevOps é crucial para o *design* de aplicações, seja para sistemas legados, produtos *greenfield* ou *brownfield*, projetos de plataforma, melhorias de recursos ou correções. Essa abordagem é relevante tanto para aplicações corporativas de três camadas quanto para produtos multicamadas, independentemente de os desenvolvedores utilizarem metodologias Agile, Waterfall ou processos ITIL. “Sem boas práticas de *design*, uma implementação DevOps não tem esperança de cumprir sua promessa de acelerar a inovação com alta qualidade e escala” (Hornbeek, 2019).

Em todos os casos de *design* de aplicação, os *designers* enfrentam desafios complexos, sendo esperado que equilibrem objetivos conflitantes. A

pressão recai sobre eles para reduzir rapidamente o atraso no trabalho, produzindo produtos de qualidade que evitem custos de rejeição e retrocessos. Além disso, há a necessidade de aumentar a porcentagem de esforço dedicado ao conteúdo criativo em detrimento do corretivo, tudo isso dentro de limitações de tempo e recursos (Wolff, 2016).

Esses objetivos conflitantes representam responsabilidades significativas, e sem o suporte das práticas de engenharia recomendadas, a experiência geral do *design* pode tornar-se um caldeirão de estresse, resultando em exaustão.

Compreender integralmente os casos de uso do cliente é imperativo para os *designers*. Características como usabilidade, confiabilidade, escalabilidade, disponibilidade, testabilidade e capacidade de suporte superam a importância de recursos individuais. Qualidade em detrimento de quantidade é a abordagem recomendada. Projetos que antecipam o uso real do cliente tendem a ser mais bem-sucedidos (Newman, 2015).

A cultura organizacional deve proporcionar apoio aos *designers*. Líderes desempenham um papel crucial ao lhes oferecer motivação, orientação e treinamento. É irrealista esperar que cada *designer* tenha conhecimento abrangente em todas as áreas, sendo aceitável que cometam erros, desde que aprendam com eles e implementem melhorias de forma ágil. Práticas de DevOps, como monitoramento contínuo e rápida remediação, exemplificam boas práticas para minimizar o impacto de eventuais erros.

As práticas de codificação de *design* assumem uma importância crítica em diversas dimensões. Produtos são concebidos para suportar embalagens modulares independentes, testes e lançamentos, resultando em particionamento em módulos com dependências mínimas entre eles. Essa abordagem possibilita a construção, o teste e o lançamento de módulos de forma isolada, sem a necessidade de compilar, testar e lançar o produto inteiro de uma vez. Segundo Newman (2015), a arquitetura dos aplicativos segue a orientação de ser modular e imutável, adotando princípios de aplicativos não monolíticos e imutáveis de 12 fatores. Essa estratégia favorece a construção de microsserviços prontos para implantação em ambientes de nuvem, em contraposição às arquiteturas monolíticas mutáveis.

As alterações no código de *software* passam por verificações prévias por meio de revisões de código por pares antes de serem comprometidas com o *5gente* de integração/tronco. O processo inclui a integração das alterações em um ambiente privado, juntamente com a versão mais recente do *5gente* de integração, seguida de testes funcionais antes de submeter às alterações ao *5gente* de integração/tronco. Adotar as seguintes práticas de engenharia recomendadas pode melhorar a experiência de *design* e a qualidade dos produtos, simplificando assim o processo no pipeline de desenvolvimento.

1.2 Quando não usar

A abordagem DevOps pode não ser apropriada em certos cenários, como em projetos pequenos e simples, em que a introdução de práticas DevOps pode ser considerada excessiva. Além disso, em situações de aplicações temporárias ou de curto prazo, sem a necessidade de manutenção contínua, a implementação completa do ciclo de vida DevOps pode ser vista como desnecessária e dispendiosa.

Outros fatores a considerar incluem a falta de comprometimento organizacional, especialmente em organizações com culturas resistentes à mudança ou sem apoio para abordagens ágeis e colaborativas, o que pode resultar em resistência à implementação do DevOps. Setores altamente regulamentados, como saúde, finanças ou governamentais, enfrentam desafios específicos ao implementar práticas DevOps em razão de suas restrições rigorosas de conformidade. A maturidade técnica da equipe de desenvolvimento e da infraestrutura de TI também é crucial, pois a falta de habilidades necessárias ou infraestrutura inadequada pode dificultar a implementação eficaz de práticas avançadas de DevOps. Além disso, restrições orçamentárias significativas podem limitar a alocação de recursos necessários para ferramentas e treinamento relacionados ao DevOps, tornando sua implementação inviável em determinados contextos. Projetos com mudanças mínimas na infraestrutura ou dependência de sistemas legados podem não tirar pleno proveito das práticas DevOps, especialmente quando essas metodologias se destacam em ambientes dinâmicos e de rápida evolução. Em resumo, a decisão de não usar DevOps dependerá de uma avaliação cuidadosa das características específicas de cada projeto e

organização, considerando o contexto, as necessidades e a cultura organizacional.

Segundo Hornbeek (2019), muitas iniciativas de DevOps resultaram em becos sem saída, mesmo com a excelência de pessoas, processos e tecnologia envolvidos. Como mencionado anteriormente, os aplicativos que não se beneficiam de prazos de entrega mais rápidos, que têm custos inadequados para justificar o investimento em DevOps ou que são demasiadamente inflexíveis para aceitar mudanças não são candidatos ideais para implementações DevOps.

> Em uma experiência com um dos meus clientes, juntamente com vinte de seus gerentes de TI e executivos seniores, estavam concluindo um projeto de avaliação de DevOps de quatro semanas. No último dia da reunião para discutir os resultados da avaliação, um executivo entrou na sala e questionou: “Qual aplicativo você avaliou para DevOps?” O gerente do projeto anunciou com orgulho o nome do produto escolhido para a avaliação. O executivo ficou visivelmente frustrado e exclamou: “Você não sabia que aquele produto já está programado para ser descontinuado neste trimestre!” Uma ação de acompanhamento foi atribuída ao gerente do projeto para considerar se avaliar um método mais adequado para a aplicação seria mais benéfico. Entretanto, durante esse período, ouvi que a mesma organização tinha um novo aplicativo crítico que enfrentou um atraso de lançamento no mercado de dez meses. Não posso deixar de especular se o resultado teria sido diferente se a aplicação mais apropriada tivesse sido avaliada em vez daquela inicialmente escolhida. (Hornbeek, 2019)

Em uma experiência com um cliente, 20 gerentes de TI e executivos seniores concluíram um projeto de avaliação de DevOps de quatro semanas. Durante a discussão dos resultados, um executivo expressou frustração ao descobrir que o produto avaliado estava programado para ser descontinuado. Isso levou a uma ação de acompanhamento para considerar métodos mais adequados, o que é típico de DevOps e Agilismo, ou seja, tem de se ter em mente sempre uma melhoria constante.

1.3 Como implementar um processo com DevOps

DevOps trata basicamente de como implementar ferramentas e cultura. Nesse sentido, Rosa (2016) propõe os passos descritos na Figura 2 a seguir.

Figura 2 – Proposta de seis passos para adoção do DevOps

Crédito: Ribkhan/Shutterrstock.

O primeiro passo consiste em delinear um projeto de desenvolvimento de *software* que apresente características propícias para a aplicação de métodos ágeis de gerenciamento de projetos. Devemos buscar projetos de baixo risco para a organização, especialmente aqueles voltados para o desenvolvimento de sistemas *web* ou *mobile*, pois se alinham de forma mais eficaz a essas metodologias (Muniz, 2019).

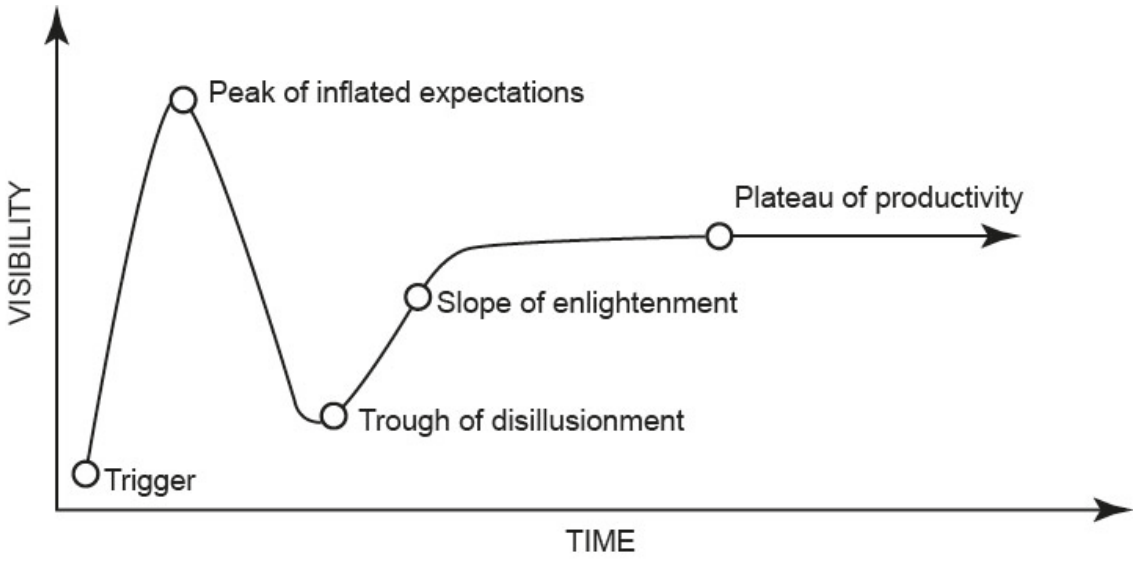
Em seguida, é crucial formar um time inicial para o projeto, composto por membros dispostos a aceitar mudanças inerentes a uma nova abordagem de trabalho. É preciso escolher pessoas que compreendam o valor da iniciativa, promovendo práticas colaborativas, comunicação eficiente e confiança. Não devemos subestimar o impacto cultural da mudança; é essencial compor uma equipe multidisciplinar, envolvendo profissionais com experiência em gestão ágil de projetos, integração contínua e testes de aplicações, além de membros da equipe operacional familiarizados com automação e integração.

O terceiro passo requer que todos os membros compreendam integralmente o significado e o propósito do DevOps. É importante alinhar a equipe em relação aos princípios do DevOps, incentivando uma mudança cultural. Devemos utilizar esse time como promotores do novo modelo de trabalho, disseminando a cultura por toda a área de TI e a organização. Essa abordagem favorece a obtenção de resultados e benefícios relacionados ao

DevOps, como maior agilidade e flexibilidade para atender às demandas do negócio.

Entender a governança e os processos de TI é crucial para antecipar possíveis obstáculos que possam limitar iniciativas e benefícios do DevOps. A recomendação é se familiarizar com os processos de controle de mudanças, configuração, atualização, incidentes e problemas, bem como com as regras e processos relacionados à comunicação, auditorias e ao gerenciamento do projeto, obtendo o apoio da alta gestão de TI e, assim, evitando barreiras que possam prejudicar as atividades e as iniciativas do DevOps. Segundo Freeman e Pryce (2012), a necessidade de convencimento de equipe e de clientes, inclusive com o uso de *personas* e técnicas ágeis é necessário (Figura 3).

Figura 3 – Propostas de Freeman e Pryce para adoção de DevOps, iniciando com um gatilho (*trigger*), passando por um momento de altas expectativas (*peak of inflated expectations*), depois por decepções (*trough of disillusionment*), por incentivos (*slope of enlightnment*) e, por fim, a produtividade (*plateau of productivity*)



Fonte: Freeman e Pryce, 2012.

As práticas do DevOps pressupõem uma colaboração aprimorada e uma mudança no foco da TI para o produto final. Devemos estabelecer metas individuais e optar por metas e objetivos atribuíveis a toda a equipe. É fundamental criar formas de medir o alcance desses objetivos ao longo do projeto, estabelecendo uma base de comparação entre o novo modelo de trabalho e o modelo tradicional da TI.

Finalmente, a automação contínua é essencial para a facilidade e simplicidade da escalabilidade na nuvem. É importante orquestrar e automatizar os processos de testes e *deploy*, minimizando a necessidade de intervenção humana, especialmente no processo de implementação do *software* nos ambientes. Ainda, devemos buscar ferramentas que facilitem a integração contínua, pois a automação aprimorará os resultados relacionados ao DevOps, eliminando erros humanos e proporcionando maior agilidade na entrega do *software* em produção.

TEMA 2 – EXPLICANDO CI/CD

2.1 Definições de CI

Na engenharia de *software*, CI é um processo antigo, que trata da prática de mesclar todas as cópias de trabalho dos desenvolvedores em uma linha principal compartilhada, várias vezes ao dia (Muniz et al., 2021).

A CI automatiza a integração de alterações de código de vários desenvolvedores em um único repositório central. Seu objetivo principal é identificar e resolver problemas de integração precocemente, antes que eles se tornem grandes e difíceis de corrigir.

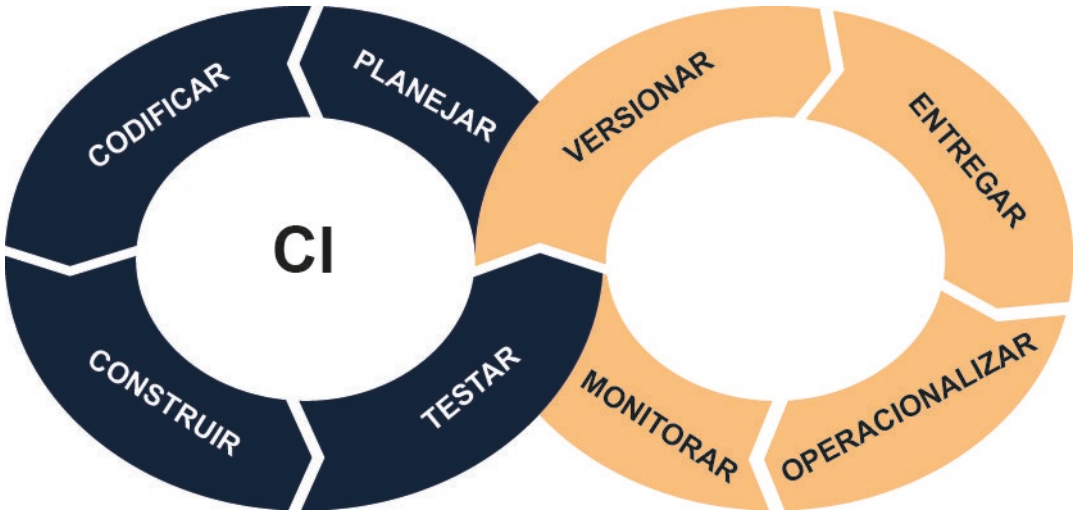
Ainda, ajuda a identificar e corrigir problemas de integração precocemente, antes que eles se tornem grandes e difíceis de corrigir. Além disso, pode ajudar a reduzir o tempo de desenvolvimento, pois os desenvolvedores não precisam esperar que as alterações de outros desenvolvedores sejam integradas antes de continuar o trabalho, o que resulta em mais tempo de desenvolvimento. Por fim, também pode ajudar a melhorar a comunicação entre os desenvolvedores, pois eles podem ver as alterações de outros desenvolvedores assim que são feitas, o que é uma prática essencial para equipes de desenvolvimento de *software* de todos os tamanhos. Em resumo, ela pode ajudar a melhorar a qualidade do código, reduzir o tempo de desenvolvimento e melhorar a comunicação entre os desenvolvedores.

A seguir, estão alguns exemplos de como a CI pode ser usada.

- Uma equipe de desenvolvimento de *software* pode usar a CI para automatizar o processo de compilação, testes e verificação de qualidade do código antes de cada lançamento.

- Uma empresa pode usar a CI para automatizar o processo de *deploy* de *software* para produção.
- Um projeto de código aberto pode usar a CI para facilitar a colaboração entre desenvolvedores de todo o mundo.
- A CI pode ser implementada usando uma variedade de ferramentas e serviços. Algumas das ferramentas de CI mais populares incluem Jenkins, Bamboo e Travis CI.

Figura 4 – Processos envolvendo CI



Fonte: Ferste, 2024.

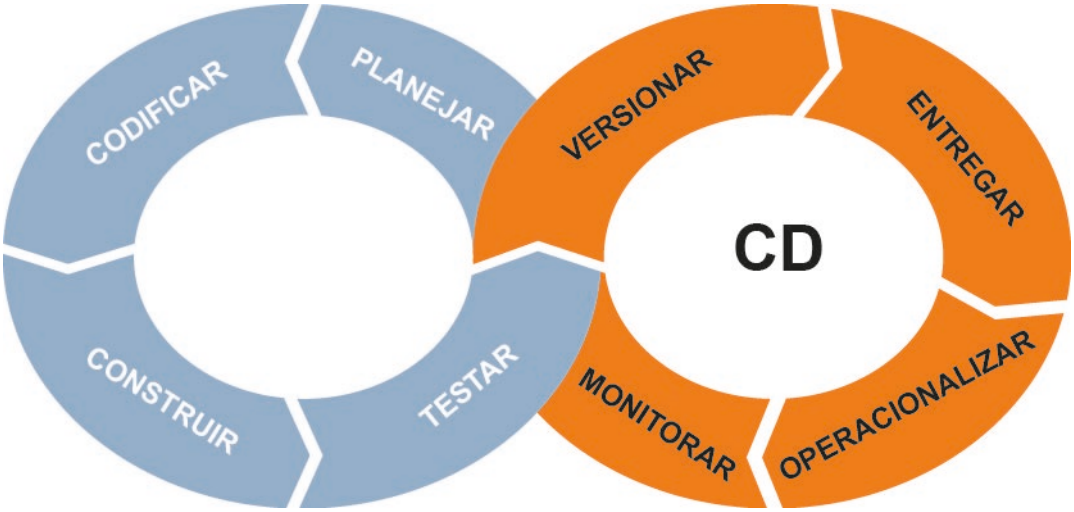
2.2 Definições de CD

Após a automação das compilações e dos testes de integração e unidade na CI, a Entrega Contínua automatiza o lançamento do código validado em um repositório. Portanto, para estabelecer um processo eficiente de entrega contínua, é crucial que a CI esteja integrada ao *pipeline* de desenvolvimento. O propósito da entrega contínua é manter a base de códigos constantemente preparada para ser implantada em um ambiente de produção (Muniz et al., 2021).

Cada fase da entrega contínua, desde a consolidação das alterações de código até a entrega de compilações prontas para produção, envolve a automação do lançamento de códigos e testes. Ao finalizar esse processo, a equipe de operações pode implantar rapidamente e com facilidade uma aplicação no ambiente de produção.

Já é desafiador para os desenvolvedores de *software* criar código funcional em suas máquinas. No entanto, mesmo quando esse objetivo é alcançado, há um longo percurso até que o *software* realmente gere valor, uma vez que este só proporciona valor quando está em ambiente de produção. A base da filosofia de entrega de *software* é desenvolver programas de modo que estejam constantemente prontos para serem implementados em produção. Esse conceito é conhecido como Entrega Contínua, pois estamos continuamente executando um *pipeline* de implantação que verifica se o *software* está em condições de ser entregue (Humble; Farley , 2010).

Figura 5 – Processos envolvendo CD



Fonte: Ferste, 2024.

O processo de entrega de *software* consiste em levar um novo recurso de desenvolvimento para produção, que, em tempos passados, consumia meses e, atualmente, pode ser concluído em minutos. Essa redução no tempo de entrega foi impulsionada por várias iniciativas, como o desenvolvimento ágil, que preconiza tempos de ciclo curtos e *feedback* rápido. A prática de integração contínua incentiva a integração diária do trabalho de todos os membros da equipe de desenvolvimento. A abordagem DevOps promove a colaboração entre desenvolvedores e operações. A infraestrutura como código possibilita provisionar novos servidores rapidamente. Juntamente, a entrega contínua mantém o produto em um estado liberável, possibilitando a rápida liberação de recursos (Chacon; Straub, 2014).

Os benefícios da entrega rápida são significativos, incluindo a geração mais ágil de valor, o aumento do retorno do investimento na construção do

recurso e o fornecimento rápido de *feedback* para o desenvolvimento futuro. Em conclusão, a entrega rápida de *software* é uma prática fundamental para equipes de desenvolvimento de *software* de todos os tamanhos, auxiliando na geração ágil de valor, no aumento do retorno do investimento e na rápida obtenção de *feedback* para o desenvolvimento futuro.

TEMA 3 – FERRAMENTAS E *FRAMEWORKS* PARA CI/CD

A adoção de ferramentas e *frameworks* para CI/CD é fundamental no cenário de desenvolvimento de *software* moderno em razão de diversos motivos que impactam positivamente o processo de entrega. Essas soluções proporcionam uma automação eficiente, eliminando tarefas manuais suscetíveis a erros e acelerando o ciclo de vida do desenvolvimento. A consistência e padronização proporcionadas por essas ferramentas garantem uma abordagem uniforme em todas as etapas do desenvolvimento, contribuindo para a qualidade e confiabilidade do *software*. A capacidade de realizar testes automatizados de forma integrada é um aspecto crucial, assegurando que novas implementações não comprometam funcionalidades existentes. Além disso, a automação facilita a implantação segura e confiável em ambientes de produção, reduzindo riscos e promovendo a consistência nas implementações.

A colaboração entre equipes de desenvolvimento e operações é otimizada com a utilização dessas ferramentas, de modo a promover uma cultura de integração contínua e entrega contínua. A rastreabilidade completa do código até a produção proporciona visibilidade sobre o *status* do *pipeline* de entrega, o que possibilita uma identificação rápida de problemas e gargalos.

Essas ferramentas também favorecem a rápida correção de problemas, uma vez que proporcionam *feedback* imediato aos desenvolvedores. Em síntese, a utilização de ferramentas e *frameworks* para CI/CD é imperativa para atender às demandas de um ambiente ágil, garantindo uma entrega de *software* eficiente, consistente e de alta qualidade.

3.1 Ferramentas de CI

A Integração Contínua é uma abordagem de desenvolvimento de *software* em que as alterações de código são frequentemente integradas em

um repositório compartilhado. O objetivo é detectar e corrigir problemas de integração rapidamente, garantindo que o código seja sempre funcional e estável. A automação desempenha um papel crucial na execução de tarefas como compilação, testes e implantação.

São ferramentas para Integração Contínua:

- Jenkins – é uma ferramenta de automação de código aberto amplamente utilizada para automação de compilação, testes e implantação;
- Travis CI – é uma plataforma de integração contínua baseada em nuvem, que se integra facilmente com repositórios no GitHub;
- CircleCI – é uma plataforma de CI/CD que automatiza o processo de construção, teste e implantação;
- GitLab CI/CD – integrado diretamente ao GitLab, fornece recursos integrados de CI/CD dentro da plataforma GitLab;
- TeamCity – desenvolvido pela JetBrains, oferece uma solução robusta para automação de compilação e integração contínua;
- Bamboo – desenvolvido pela Atlassian, é uma ferramenta de CI/CD que se integra bem com outros produtos Atlassian, como Jira e Bitbucket;
- GitHub Actions – é uma plataforma integrada diretamente ao GitHub para automação de fluxos de trabalho, incluindo CI/CD;
- Drone – é uma plataforma de CI/CD leve e flexível que pode ser executada como um contêiner Docker.

É importante nos lembrarmos de que a escolha da ferramenta de CI depende dos requisitos específicos do projeto, da preferência da equipe e da integração com outras ferramentas no ecossistema de desenvolvimento.

3.2 Ferramentas de CD

Seguem exemplos de ferramentas que se encaixam no contexto de entrega contínua:

- Docker: Embora o Docker não seja estritamente uma ferramenta de Continuous Delivery, é uma parte fundamental do processo de entrega moderna. Ele é usado para encapsular aplicativos em contêineres, garantindo que eles sejam executados de maneira consistente em diferentes ambientes. No contexto de CD, os contêineres Docker podem ser

implantados de forma rápida e confiável em ambientes de teste, de pré-produção e de produção, tornando o processo de entrega mais eficiente e consistente.

- AWS (Amazon Web Services): A AWS oferece uma ampla gama de serviços que podem ser integrados ao processo de Continuous Delivery. Por exemplo, o AWS CodePipeline pode ser usado para automatizar a construção, teste e implantação de aplicativos. O AWS CodeDeploy pode ser usado para implantar aplicativos em instâncias EC2 ou em serviços gerenciados, como o Amazon ECS (Elastic Container Service) ou o Amazon EKS (Elastic Kubernetes Service). Além disso, a AWS oferece uma variedade de serviços para monitoramento, como o Amazon CloudWatch, que pode ser integrado ao processo de CD para fornecer insights sobre o desempenho do aplicativo.
- Chef: Trata de automação de infraestrutura que pode ser usada para provisionar e configurar servidores de forma automatizada. No contexto de Continuous Delivery, o Chef pode ser usado para garantir que os ambientes de teste, de pré-produção e de produção sejam configurados de maneira consistente e replicável. Ele pode ser integrado aos pipelines de CD para garantir que as configurações de infraestrutura sejam atualizadas automaticamente junto com o código do aplicativo.
- Azure: Assim como a AWS, a Microsoft Azure oferece uma variedade de serviços que podem ser integrados ao processo de Continuous Delivery. O Azure DevOps oferece recursos para automatizar a construção, teste e implantação de aplicativos, enquanto o Azure Kubernetes Service (AKS) pode ser usado para implantar e gerenciar aplicativos em contêineres Kubernetes. Além disso, o Azure Monitor fornece recursos avançados de monitoramento e alerta que podem ser integrados ao processo de CD para garantir que os aplicativos sejam entregues com alta qualidade e desempenho.
- Nagios: Nagios é uma ferramenta de monitoramento de infraestrutura de código aberto que ajuda a detectar e resolver problemas de rede e sistemas. Embora não seja diretamente uma ferramenta de Continuous Delivery, o Nagios pode ser integrado aos pipelines de CD para fornecer feedback em tempo real sobre a integridade e o desempenho dos sistemas durante o processo de entrega.

- Splunk: plataforma de análise de dados que permite coletar, indexar e visualizar logs e métricas de aplicativos e infraestrutura. No contexto de Continuous Delivery, o Splunk pode ser usado para monitorar e analisar o desempenho dos aplicativos em tempo real, identificando problemas e gargalos que podem afetar a entrega.
- Terraform: uma ferramenta de infraestrutura como código que permite provisionar e gerenciar recursos de infraestrutura de forma automatizada e declarativa. No contexto de Continuous Delivery, o Terraform pode ser usado para definir e versionar a infraestrutura necessária para executar aplicativos, garantindo que os ambientes de implantação sejam consistentes e replicáveis em todos os estágios do pipeline de entrega.

Essas são apenas algumas das maneiras pelas quais essas ferramentas podem ser utilizadas no contexto de Continuous Delivery. Cada uma delas oferece recursos e funcionalidades únicas que podem ajudar as equipes de desenvolvimento a automatizar e otimizar o processo de entrega de software.

TEMA 4 – *SCRIPTS* DE CONSTRUÇÃO E AUTOMAÇÃO

4.1 *Pipeline* de CI/CD

Uma *pipeline* em DevOps refere-se a uma série automatizada de processos que facilitam o desenvolvimento, o teste e a implantação contínua de *software*. Essa abordagem é conhecida como “Integração Contínua e Entrega Contínua” (CI/CD), sendo uma prática essencial em DevOps. A *pipeline* é uma representação visual e automatizada do fluxo de trabalho do desenvolvimento de *software*, que envolve várias etapas, desde a escrita do código até a entrega do produto final. Cada etapa da *pipeline* é um estágio no ciclo de vida do desenvolvimento de *software* e pode incluir tarefas como compilação, testes automatizados, revisões de código, empacotamento, implantação e monitoramento.

Os principais componentes de uma *pipeline* em DevOps incluem:

- Para Integração Contínua (CI), visto na Figura 3 anterior:
  - planejar: gerenciar o fluxo de mudanças;
  - codificar: compilar o código-fonte para gerar executáveis ou artefatos;

  - construir: basicamente realizar entregas ou *builds* automatizadas entre o ambiente local de desenvolvimento e algum servidor central;
  - testar: executar testes automatizados para garantir a integridade do código. Avaliação automática ou manual do código por meio de revisões de código.
- Para Entrega Contínua (CD), visto na Figura 4 anterior:
  - versionar: preparar o *software* para implantação, como criar pacotes ou contêineres;
  - entregar: implantar automaticamente o *software* em ambientes de teste e/ou produção;
  - operacionalizar: tornar viável a entrega em produção ou algum ambiente determinado;
  - monitorar: monitorar a aplicação para identificar problemas ou melhorias.

A automação de *pipelines* em DevOps oferece vários benefícios, incluindo:

- rápida detecção de problemas – testes automáticos e revisões de código ajudam a identificar problemas rapidamente;
- entrega contínua – facilita a entrega frequente e consistente de *software*;
- redução de erros – a automação reduz o risco de erros humanos e melhora a qualidade do *software*;
- *feedback* imediato – desenvolvedores recebem *feedback* rápido sobre suas alterações.

As ferramentas de automação, como Jenkins, GitLab CI, Travis CI, GitHub Actions, entre diversas outras, são comumente usadas para criar e gerenciar *pipelines* em DevOps. Essas ferramentas possibilitam a configuração, a execução e o monitoramento automatizados de cada estágio da *pipeline*.

4.2 Criação de um exemplo de *pipeline* de CI/CD

Em um *pipeline* DevOps, os *scripts* de construção e automação são essenciais para garantir a entrega contínua e a integração contínua do *software*. A seguir, apresentamos um exemplo básico de um *pipeline* DevOps

usando Jenkins, um popular servidor de automação de código aberto, com *scripts* de construção em diferentes estágios.

4.2.1 *Pipeline* DevOps com Jenkins e *Scripts*

- **Configuração inicial:**
  - configurar um novo *pipeline* no Jenkins;
  - conectar o Jenkins ao repositório de código-fonte.
- ***Script* de construção (*Jenkinsfile*):** o *Jenkinsfile* é um arquivo de configuração de *pipeline* que pode residir no repositório de código. A seguir, apresentamos um exemplo básico de um *pipeline*. Nesse exemplo, o *pipeline* tem quatro estágios – "*Checkout*", para obter o código, "*Build*", para compilar o código, "*Test*", para executar testes automatizados, e "*Deploy*", para implantar o aplicativo.

```
Pipeline {
    17gente any

    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Build') {
            steps {
                script {
                    // Script de construção, por exemplo,
compilar código
                    sh 'make clean build'
                }
            }
        }

        stage('Test') {
            steps {
                script {
                    // Script de teste, por exemplo, executar
testes automatizados
                    sh 'make test'
                }
            }
        }
```

```
        stage('Deploy') {
            steps {
                script {
                    // Script de implantação, por exemplo,
implantar em um ambiente de produção
                    sh 'make deploy'
                }
            }
        }
    }
}
```

- *Scripts* de Construção Reais (*Makefile*): os *scripts* reais de construção podem residir em arquivos como *Makefile*. A seguir, apresentamos um exemplo simples.

```
Clean:
    rm -rf ./build

build:
    mkdir -p ./build
    gcc -o ./build/my_program source/*.c

test:
    ./build/my_program -test

deploy:
    # Lógica de implantação, por exemplo, copiar arquivos para
um servidor
```

Esses *scripts* são chamados no *Jenkinsfile* nos estágios apropriados. Considerações adicionais:

- certificar-se de ter as ferramentas necessárias (como *make*, compiladores) instaladas nas máquinas executoras do Jenkins;
- configurar credenciais para acessar ambientes de implantação.

Esse é um exemplo básico, e a complexidade pode aumentar dependendo das necessidades do projeto. É importante adaptarmos os *scripts* e o *pipeline* conforme necessário para atender aos requisitos específicos do nosso projeto e ambiente.

TEMA 5 – TESTES AUTOMATIZADOS E GARANTIA DE QUALIDADE

Para Pressman e Maxim (2014), a qualidade de *software* é definida como “conformidade com requisitos funcionais e de desempenho

explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas, que são esperadas em todo *software* desenvolvido profissionalmente".

5.1 Métricas

A medição da qualidade em DevOps envolve a avaliação de diversos aspectos ao longo do ciclo de vida do desenvolvimento e operações. A seguir, estão algumas práticas e métricas que podem ser usadas para medir a qualidade em ambientes DevOps.

- Taxa de entrega: número de releases ou *deploys* bem-sucedidos por unidade de tempo. Indica a frequência com que novas funcionalidades ou correções são entregues ao ambiente de produção.
- Tempo de ciclo: tempo necessário para levar uma alteração do código até a produção. Mede a eficiência do processo de desenvolvimento e *deployment*.
- Taxa de sucesso de testes automatizados: porcentagem de testes automatizados bem-sucedidos em relação ao total. Indica a confiabilidade dos testes automatizados, garantindo uma detecção precoce de problemas.
- Tempo de recuperação (MTTR – *Mean Time to Recovery*): tempo médio necessário para restaurar serviços após uma falha. Avalia a eficácia na resolução de problemas e a recuperação rápida de falhas.
- Taxa de incidência de defeitos (*Defect Rate*): número de defeitos identificados em produção em relação ao número total de alterações. Ajuda a entender a estabilidade do sistema após as implementações.
- Qualidade do código: avaliação da qualidade do código-fonte por meio de ferramentas de análise estática. Indica a conformidade do código com padrões de codificação e boas práticas.
- Tempo de execução de testes: tempo necessário para executar testes automatizados. Avalia a eficiência dos testes e pode indicar possíveis gargalos no processo.
- Satisfação do cliente: pesquisas de satisfação do cliente ou *feedback* direto. Reflete a experiência do usuário e a qualidade percebida pelo cliente.

- Taxa de rejeição de *builds*: número de *builds* rejeitadas em relação ao total. Indica a qualidade das alterações integradas ao repositório principal.
- Monitoramento de *performance* em produção: métricas de desempenho e estabilidade em ambiente de produção. Ajuda a identificar degradações de desempenho ou problemas em tempo real.

É importante adaptar essas métricas às necessidades específicas do projeto e da organização. Além disso, a automação é fundamental para coletar e analisar essas métricas de forma eficiente ao longo do ciclo de vida DevOps.

5.1 Testes

A medição da qualidade em DevOps envolve a avaliação de diversos aspectos ao longo do ciclo de vida do desenvolvimento e operações. Aqui estão algumas práticas e métricas que podem ser usadas para medir a qualidade e testes.

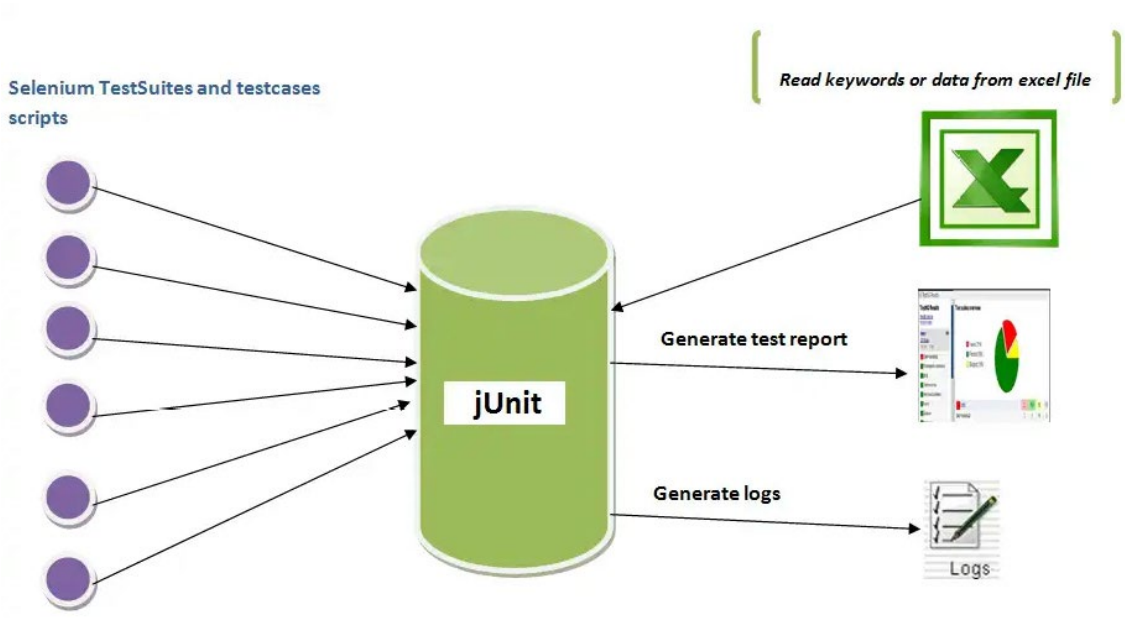
Em ambientes DevOps, uma diversidade de testes é aplicada para assegurar a qualidade do *software* ao longo de todo o ciclo de desenvolvimento. Isso inclui os testes unitários, que avaliam a funcionalidade de unidades individuais de código, e os testes de integração, que verificam a interação harmoniosa entre diferentes componentes do sistema. Os testes de aceitação do usuário (UAT) garantem a conformidade do sistema com os requisitos específicos do usuário, enquanto os testes de desempenho avaliam sua eficiência e escalabilidade. Os testes de segurança são implementados para identificar possíveis vulnerabilidades, e os testes de regressão garantem que novas alterações não introduzam falhas em funcionalidades existentes. Os testes de regressão verificam se as funcionalidades-chave estão operacionais após uma alteração, oferecendo uma rápida verificação inicial. Adicionalmente, os testes de usabilidade avaliam a facilidade de uso, e os testes de registros e *logs* garantem que as informações relevantes sejam registradas conforme o esperado. Essa abordagem abrangente de testes em ambientes DevOps assegura a qualidade contínua do *software*, integrando avaliações em todo o processo de desenvolvimento e implementação (Hornbeek, 2019).

As atividades de testes são realizadas sempre no final de cada versão, sendo esse o momento em que a equipe de testes analisa os requisitos com

base em documentações geradas. As equipes de testes tinham uma burocracia em suas atividades: criação de planos, geração de documentos, criação de cenários de testes, relatórios de execução. Ou seja, uma certa autonomia em suas atividades, mas um alto grau de dependência da equipe de desenvolvimento, na qual suas atividades apenas iniciavam após todo o processo de desenvolvimento encerrado.

Testes de unidade são realizados em um nível de granularidade muito baixo, próximo ao código-fonte do aplicativo. Eles envolvem a verificação de métodos e funções individuais de classes, componentes ou módulos utilizados pelo *software*. Esses testes, geralmente de baixo custo para automação, podem ser executados rapidamente por um servidor de integração contínua.

Figura 6 – *Framework* de Testes Junit



Fonte: Digital Innovation One.

Os testes de integração, por sua vez, garantem a adequada interação entre diferentes módulos ou serviços utilizados pelo aplicativo. Exemplos incluem a verificação da interação com o banco de dados ou a garantia de que os microsserviços operem conjuntamente conforme o esperado. Esses testes, embora mais onerosos, exigem que várias partes do aplicativo estejam ativas e em execução.

Os testes funcionais concentram-se nos requisitos de negócios da aplicação, verificando apenas a saída de uma ação, sem avaliar os estados intermediários do sistema durante a execução dessa ação. Diferentemente dos

testes de integração, eles se concentram na conformidade com os requisitos estabelecidos.

Os testes de ponta a ponta replicam o comportamento do usuário em um ambiente de aplicativo completo, verificando se os diversos fluxos de usuário funcionam conforme o esperado. Embora sejam muito úteis, esses testes têm um custo elevado e pode ser desafiador mantê-los quando automatizados.

Testes de aceitação são formais e avaliam se um sistema atende aos requisitos de negócios, exigindo que todo o aplicativo esteja ativo e em execução. Eles replicam os comportamentos do usuário e podem incluir a medição do desempenho do sistema, rejeitando alterações se determinadas metas não forem atendidas.

Testes de desempenho avaliam o desempenho de um sistema sob uma carga de trabalho específica, medindo confiabilidade, velocidade, escalabilidade e capacidade de resposta. Esses testes ajudam a identificar gargalos, medir a estabilidade durante picos de tráfego e determinar se um aplicativo atende aos requisitos de desempenho.

Os testes de fumaça, ou testes de sanidade, são básicos e verificam a funcionalidade essencial do aplicativo. Realizados de maneira rápida, sua meta é garantir que os principais recursos do sistema estejam funcionando conforme o esperado, sendo úteis após a implementação de um novo *build* ou durante a fase inicial de testes.

5.2 Qualidade

A garantia de qualidade representa “o conjunto de atividades sistemáticas realizadas para assegurar a adequação do artefato como um todo ao seu propósito de uso”. Sob essa perspectiva, a garantia da qualidade de *software* engloba diversas ações destinadas a avaliar o nível de qualidade presente no produto analisado, certificando-se de que atenda às especificações explícitas e implícitas. Essa abordagem depende de fatores-chave, incluindo o processo de desenvolvimento empregado, a participação das pessoas no projeto e a execução de atividades específicas voltadas para garantir a qualidade.

Aí temos uma grande vantagem para o DevOps: como tudo é automatizado, é muito fácil tratar de indicadores de desempenho e qualidade,

pois, se bem feito, já vai gerar automaticamente um painel, conforme Figura 7 a seguir.

Figura 7 – *Dashboard* ou painel com indicadores

Crédito: ZinetroN/Shutterstock.

Embora o Terraform não inclua comandos específicos para medição de qualidade, podemos utilizar uma combinação de boas práticas e ferramentas externas para garantir a qualidade do código Terraform. A seguir, estão algumas sugestões.

- Validação de sintaxe: executar o comando *terraform validate* para verificar a sintaxe do código Terraform.

  Terraform validate

- Formatação consistente: usar o comando *terraform fmt* para formatar o código de acordo com as convenções de estilo recomendadas.

  Terraform fmt

- Verificação estática: ferramentas como *tflint* podem ser úteis para verificação estática do código Terraform em busca de possíveis problemas ou violações de boas práticas.

  Tflint

- Auditoria de segurança: considerar a utilização de ferramentas de análise de segurança, como tfsec, para identificar possíveis vulnerabilidades e melhorar a postura de segurança do código Terraform.

  Tfsec

- Testes: implementar testes de integração usando ferramentas como *kitchen-terraform* para validar se a infraestrutura é provisionada conforme esperado.

  Kitchen test

- Automação e CI/CD: integrar os comandos mencionados anteriormente em seus *pipelines* de CI/CD para garantir que a validação, a formatação e outros controles de qualidade sejam executados automaticamente antes da aplicação de alterações no ambiente.

Ao adotar essas práticas e ferramentas, podemos melhorar a qualidade do nosso código Terraform, garantindo que ele seja legível, consistente, seguro e funcione conforme o esperado.

FINALIZANDO

O CI/CD *pipeline*, que representa a prática de Integração Contínua (CI) e Entrega Contínua (CD) no desenvolvimento de *software*, é de extrema importância na busca por eficiência e confiabilidade nos processos de entrega de *software*. A Integração Contínua envolve a integração frequente de alterações no código fonte, o que possibilita uma detecção precoce de possíveis conflitos e erros, promovendo uma colaboração suave entre membros da equipe. A Entrega Contínua, por sua vez, concentra-se na automação do processo de implantação, possibilitando a liberação rápida e consistente de novas versões do *software*.

A qualidade desempenha um papel crucial nesse contexto, sendo intrínseca a cada etapa do CI/CD *pipeline*. Automatizar a construção do *software* e a execução de testes é essencial para garantir que as alterações introduzidas não comprometam a estabilidade e a funcionalidade do sistema. Isso não somente acelera o ciclo de desenvolvimento, mas também proporciona uma garantia de qualidade contínua, contribuindo para a criação

de *software* mais robusto e confiável. A integração harmoniosa de práticas de CI/CD e a ênfase na qualidade formam a base para o desenvolvimento ágil e eficiente de *software*.

REFERÊNCIAS

CHACON, S.; STRAUB, B. **Pro Git**. 2. Ed. São Paulo: Apress, 2014.

FREEMAN, S.; PRYCE, N. **Desenvolvimento de *software* orientado a objetos, guiado por testes**. Rio de Janeiro: Alta Books, 2012.

HORNBEEK, M. **Engineering DevOps**: From Chaos to Continuous Improvement… and Beyond. Pennsauken: BookBaby, 2019.

HUMBLE, J.; FARLEY, D. **Continuous Delivery**: Reliable Software Releases through Build, Test, and Deployment Automation Glenview: Addison-Wesley Professional, 2010.

MUNIZ, A. et al. **Jornada Azure DevOps**: unindo teoria e prática com o objetivo de acelerar o aprendizado do Azure DevOps para quem está iniciando. Rio de Janeiro: Brasport, 2021.

_____. **Jornada DevOps**: unindo cultura ágil, *Lean* e tecnologia para entrega de *software* com qualidade. Rio de Janeiro: Brasport, 2019.

NEWMAN, S. **Building Microservices**: Designing Fine-Grained Systems. Sepastopol: O'Reilly & Associates Inc, 2015.

PRESSMAN, R. S.; MAXIM, B. **Software Engineering**: a practitioner's approach. 8. Ed. New York: McGraw-Hill, 2014.

ROSA, D. 6 passos para iniciar no DevOps. **Jornada Para a Nuvem**, 5 mar. 2018. Disponível em: <https://jornadaparanuvem.com.br/6-passos-para-iniciar-na-jornada-do-devops/>. Acesso em: 17 abr. 2024.

WOLFF, E. **Microservices**: Flexible Sofware Architectures. Victoria: Leanpub, 2016.